CROWD, a platform for the crowdsourcing of complex tasks

Ahmad Chettih
University of Rennes 1

David Gross-Amblard
University of Rennes 1 / IRISA
dga@irisa.fr

David Guyon
University of Rennes 1

Erwann Legeay
University of Rennes 1

Zoltán Miklós
University of Rennes 1 / IRISA
zoltan.miklos@irisa.fr

ABSTRACT

Crowdsourcing is an emerging technique that enables to involve humans into information gathering or computational tasks. With the help of crowdsourcing platforms a group of participants (workers) can solve otherwise difficult problems. While the existing, generic crowdsourcing platforms such as Amazon Mechanical Turk have been used to address various challenges, they only support simple questions that we consider as basic tasks. In this demo we present CROWD, a platform where one can submit a workflow, which is a composition of such basic tasks. CROWD also supports a simple skill-management mechanism: each basic task is annotated with expertise tags. Upon the validation of completed tasks, the worker's expertise is updated according to these tags. The worker's expertise can later be used for better task selection. CROWD is implemented in Python/Django, and can be used both on the Web or on mobile devices.

1. INTRODUCTION

Crowdsourcing is a recent technique that allows so called taskers to rely on an unknown crowd on the Internet to solve a difficult task. Many ad hoc crowdsourcing platforms are devoted to the resolution of specialized tasks: see for example the FoldIt project[1] for the discovery of new protein foldings, or Transifex[2] for the translation of technical documents. Besides, generic crowdsourcing platforms have emmerged, such as Amazon Mechanical Turk (AMT[3]), CrowdFlower[4] or CloudFactory[5] , to name a few. These generic systems support mainly simple data acquisition tasks presented as a sequence of questions (forms). They are however not suited for complex tasks that require repetitions, alternatives or conditional execution. A typical example of such complex tasks is cooperative relief–support during disasters[6] [7], where Internet-connected people assist rescue teams by providing outside information (based on e.g. satellite pictures). We use the following complex task T as a running example:

> T: A storm has just hit Miami. Please help the coordination rescue operations, by providing information on available hospitals and passable roads to reach them. If you are trained in emergency management or you have experience with road traffic control, your are very welcome to join our online support team.

This task can be submitted to existing platforms, but as a unique text block. One could imagine a more structured and and more precise formulation of the same task:

> T (rewritten): A storm has just hit Miami. Please help the coordination of rescue operations, by first (T_1) locating nearby hospitals on this map (link given). Then, (T_2) obtain their phone numbers, and then (T_3) try to contact them to obtain their availability. Meanwhile, (T_4) list the passable roads reaching these hospitals. If you are trained in emergency management or you have experience with road traffic control, your are very welcome to join our online rescue teams.

This latter formulation shows opportunity for task parallelization and expertise tagging, however such queries are not supported by the existing platforms. Even if the API of the crowdsourcing platforms enables to realize the combination of basic tasks, this would require expert programming skills and considerable efforts for each task. Instead we propose a platform where one can submit such complex workflows directly.

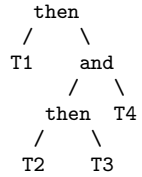
Contribution. In this demo we present CROWD, a platform where **one can submit a whole *workflow*, that is a composition of basic tasks**. The workflow is expressed as an execution plan with various composition operators.

[1] `http://fold.it`
[2] `https://www.transifex.com/`
[3] `https://www.mturk.com/`
[4] `http://www.crowdflower.com/`
[5] `http://www.cloudfactory.com`

[6] `http://www.usnews.com/opinion/articles/2012/11/23/how-to-make-crowdsourcing-disaster-relief-work-better`
[7] `http://www.nature.com/news/crowdsourcing-goes-mainstream-in-typhoon-response-1.14186`

(c) 2014, Copyright is with the authors. Published in the Proceedings of the BDA 2014 Conference (October 14, 2014, Grenoble-Autrans, France). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.
(c) 2014, Droits restant aux auteurs. Publié dans les actes de la conférence BDA 2014 (14 octobre 2014, Grenoble-Autrans, France). Redistribution de cet article autorisée selon les termes de la licence Creative Commons CC-by-nc-nd 4.0.
BDA 14 octobre 2014, Grenoble-Autrans, France.

The workflow associated with our previous example is shown below. Its interpretation is, informally, "perform T1, then perform (in parallel) T4 and the sequence T2 and T3".



In this example, basic tasks (T_1 to T_4) can be submitted to existing platforms, and parallelism between tasks can be expressed. In our model the available operators are sequences (then), alternatives (or), conjunction (and), tests (if) and loops (while).

CROWD also supports a **simple skill-management mechanism**. Each basic task can be annotated by a list of expertise tags (e.g. "emergency, hospital" for T_1 to T_3 and "road, traffic" for T_4). Upon the validation or rejection of completed tasks, the workers expertise score is updated according to the relevant tags. The worker's expertise can later be used for better task assignment and the evaluation of the available expertise on a completed execution plan.

CROWD is implemented in Python/Django, and can be used both on the Web or on mobile devices. Participation is free (but extensible with fee facilities for participants if required) and symmetric (any user can participate in or create a task, contrary to existing platforms).

The rest of this demo is organized as follows: after giving the related work in Section 2, we expose the model underlying CROWD in Section 3. Then we detail our system in Section 4 and conclude with our demo scenario in Section 5.

2. RELATED WORK

The idea of composing smaller tasks and offering a composite task execution service is on the agenda of several startup companies. For example, Ville Miettinen, the CEO of MICROTASK[8], mentioned in a recent interview: "In the long term, all human processes that can be standardized will be available as a cloud service..." [9] MICROTASK already offers some composite services, but these services are predefined.

While basic tasks can be easily deployed on Amazon mechanical turk or similar platforms, sequences of tasks have to be modeled in a form. Alternatives can be hacked in Crowdflower using a specific $only-if$ construct in their CML language. Up to our knowledge, AMT nor CrowdFlower can model for example parallel tasks within a sequence.

Recent academic works propose some form of programming languages that enable a more flexible task composition, including [1, 2]. These procedural approaches offer a fine control of human-based computation, but the required skill for the task developer is high. Several declarative approaches that ease data acquisition campaigns have also been constructed recently [5, 8, 7, 6], but they do not reason on task composition. A generic, data oriented crowd system is presented in [3]. This active rules-based system is able to implement workflows that are similar to ours, but this requires complex skills for defining appropriate triggers. Sophisticated expertise mining models have also been proposed in the context of crowdsourcing (e.g. [4]), but independently from the workflow where these tasks belong to.

[8] http://www.microtask.com/

[9] http://venturebeat.com/2011/03/22/crowdsourcing-startup-microtask-gets-gamers-to-do-some-real-work/

3. MODEL

We present in this section the model underlying CROWD. This model encompasses the workflow execution, workflows intermediate results, result provenance, user and task expertise and task validation.

Participants and Expertise. Each participant in a CROWD process is uniquely identified, with identifiers denoted as *uid* in the sequel.

We suppose given a vocabulary of expertise tags, such as "hospital" or "traffic" (the vocabulary of these tags is free. A systematic building of these tags, for example as a taxonomy, is out of the scope of this demo). The expertise E_{uid} of participant *uid* is a function mapping a expertise tag e to an expertise integer score $E_{uid}(e)$ (where 0 means no expertise at all). The initial expertise of a new participant maps all expertise tag to 0.

Basic and Complex Tasks. A user can submit tasks to the system (we then call it a tasker, but any participant can become a tasker). The concrete syntax of a *basic task* such as T_1 is for example

```
ask(2,"Give the location of nearby hospitals",
      [hospital, emergency])
```

It denotes the triggering of 2 questions on hospital locations on the CROWD platform for available users. The task will be tagged by the hospital and emergency expertise. Once 2 distinct participants have answered this question, the task is finished. The result of this task is a list of (key,value) pairs, where the key is formed by the running task identifier denoted by *tid*, the participant's *uid* and the answer number. It is noteworthy that tasks results in CROWD are not typed, but are nested lists of (key,value) pairs (for the sake of simplicity we do not elaborate here on constraints on participant answers, like requiring the answer to be an integer of to respect a regular expression. This is a classical topic for all crowdsourcing platform). Hence, a possible result for T_1, denoted $res(T_1)$ could be

```
{(tid1-uid1-1,"Miami Children hospital"),
 (tid1-uid5-2,"University of Miami hospital")}
```

In turn, a *complex task* is either a basic task or a composition of complex tasks. When several tasks are involved, the result of one task can be the input of another task (in practice, the participant of the second task can see this input). Given two complex tasks t and t', we use a prefix syntax to connect them: $then(t, t')$ (sequence),$and(t, t')$ (conjunction), $or(t, t')$ (alternative), $while(t, t')$ (loop). The conditional structure $if(t, t', t'')$ requires a third task t''.

The evaluation of $then(t, t')$ launches t, and upon termination, launches t'. The result of t is passed as input to t'. The final result is the tuple $(then, res(t), res(t'))$ (the result of t is preserved in the trace of the overall execution). The evaluation of $and(t, t')$ launches both tasks in parallel, and both must terminate for the *and* task to finish. The result is the tuple $(and, res(t), res(t'))$. The evaluation of

$or(t, t')$ launches both tasks in parallel, the first to finish, say t, interrupts the other. The result is the tuple $(or, res(t))$. The evaluation of $if(t, t', t'')$ launches the evaluation of t. Upon termination, if the first tuple of the result matches $(\_, true)$, task t' is launched. Otherwise t'' is launched. The if result is the tuple $(if, res(t))$ or $(if, res(t'))$ according to the condition output. Finally, the evaluation of $while(t, t')$ launches t and t' each time the result of t matches the tuple $(\_, true)$, and otherwise finishes. The result is the tuple $(while, res(t^*))$ where t^* is the last execution of t' in the loop.

Based on this syntax, the translation of our running example would be the following, if we expect at least 2 answers of each kind:

```
then(
    ask(2,"Give the location of nearby hospitals",
    [emergency,hospital]),
    and(
      then(
        ask(2,"Obtain phone numbers of the previous hospitals",
          [emergency,hospital]),
        ask(2*"Using the previous hospital phone numbers,
         obtain their availability",
         [emergency,hospital])
      )
      ask(2,"List passable roads reaching the
         previous hospitals",
         [road, traffic]
      )
    )
)
```

While this syntax is sufficient, we also provide an infix notation for quick task design for *ask*, *and*, *or* and *then*, using shortcuts {∗}, &&, || and ;; respectively. The corresponding notation of the previous example is the following:

```
{ 2*"Give the location of nearby hospitals"
    [emergency,hospital]
}
;;
( (
    { 2*"Obtain phone numbers of the previous hospitals"
        [emergency,hospital]
    }
    ;;
    { 2*"Using the previous hospital phone numbers,
         obtain their availability"
         [emergency,hospital]
    }
  )
  &&
    { 2*"List passable roads reaching the
         previous hospitals"
         [road, traffic]
    }
)
```

Expertise Update. Once a complex task T is finished, the tasker can see the result and all task intermediate results and provenance. Then the tasker can validate the overall task or reject it. In case of validation (resp. rejection), we set a *score* value to 1 (resp. -1). We consider a participant *uid* who contributed to T. We update this participant's expertise according to this score, for each expertise tag he contributed. More precisely, let $E_T = \{e_1, \ldots, e_n\}$ be the set of expertise tags associated with the basic tasks of T that were answered by participant *uid*. We modify the expertise of *uid* such that

$$E_{uid}(e_i) := E_{uid}(e_i) + score, \text{ for each } i = 1 \ldots, n.$$

4. SYSTEM OVERVIEW

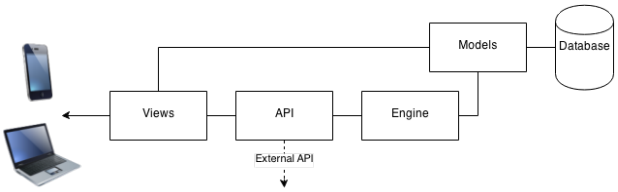


Figure 1: The CROWD Architecture

Figure 1 shows the overall architecture of the CROWD system. It is composed by several components: a web-based graphical user interface, an internal API, a database and an engine for task scheduling.

The GUI is based on the Django framework. Users can log using a registered account or anonymously. A panel shows all the available tasks ranked by expertise requirements. Another panel allows for task launching. Tasks can be entered either using the concrete syntax described in Section 3, or using a workflow design interface shown in Figure 2.

Tasks are modeled as python objects and mapped to a Postgresql database using the Django ORM module with concurrency control enabled. The classical execution cycle of the system is as follows. Each complex task is decomposed as a set of task nodes. Each root node begins in the *start* state. Then the tree of tasks is visited top-down: the left child of a sequence (*then*) node is started, and all child of a started *and* or started *or* node, and the conditions of a started if or *while* node are started. Any available basic task in the *start* state is presented on the user interface. Each time a basic task is answered by a participant, a counter is updated until the total number of awaited answers is reached. Then, the basic task turns to the $finished$ state. The termination of a child of a *or* node terminates the node. The termination of all the child node of an *and* node terminates the node. The termination of the left child of a *then* node launches the right node (similarly for f and *while* nodes).

When a complex task is fully answered, its state changes and becomes $finished$. At this point the tasker can get his answers thanks to the related web page. Results can be exported in a CSV document. The tasker can validate or reject the task, and users experience is updated accordingly.

5. DEMONSTRATION

In the demonstration, we will illustrate the following aspects of the CROWD platform: logging and task answering, basic and complex task creation and advanced XHTML features. A video of a preliminary version of our demo is available here:

`https://www.youtube.com/watch?v=3Zd6QpHpYhQ&feature=youtu.be`

Logging and Task Answering. In the first part of this demo, users are invited to log on the platform, either by defining a user account, or by logging anonymously in one click (anonymous logging is an incentive for the free participation to interesting tasks, without leaving personal traces on the system). Logging can be performed either on the website or on the user's personal mobile phone.

Users will be able to browse the available tasks, to select one (for example, "How many persons is now attending this demo ?"), and to answer it on the website or on their phones.

Basic and Complex Task Creation. In this part, users will be able to define a new task, either using the concrete syntax in an editor, or by using a workflow editor that allows to describe a tree in graphical mode (this last feature is not available on phones). After defining basic tasks, we will demonstrate the use of connectors to build a complex task. Finally, we will show how to express the required expertise for these tasks.

Advanced Features. In the last part of the demo, the Miami storm scenario is launched, allowing the audience to participate (virtually) in the relief. Participants are invited to locate hospitals on an interactive Google Map, launched from the task board. It is noteworthy that any XHTML code can be inserted in task descriptions, which allows for rich interaction with external platforms (Figure 3).

Monitoring Tasks. All along its lifetime, users can watch the progression of a complex task and of its related basic tasks using the "My Tasks" panel. By clicking on the answer link, detailed answers of each basic tasks composing the complex task are shown. An export as a csv file is available.

Expertise Ranking. We will demonstrate how the validation of a task impacts on the user's expertise, and how this expertise affects the ranking of the next proposed task.

Acknowledgements

We would like to thank the following persons for their help with the development of CROWD during their undergraduation project: Pierre-Luc Blay, Thomas Daniellou, Archibald Jego, Guillaume Landurein, Sylvain Medard, Houssam Ouazzani and Victor Petit.

6. REFERENCES

[1] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 53–64, 2011.

[2] Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor. Automan: a platform for integrating human-based and digital computation. *SIGPLAN Not.*, 47(10):639–654, October 2012.

[3] Alessandro Bozzon, Marco Brambilla, Stefano Ceri, and Andrea Mauri. Reactive crowdsourcing. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 153–164, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[4] Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Matteo Silvestri, and Giuliano Vesci. Choosing the right crowd: Expert finding in social networks. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 637–648, New York, NY, USA, 2013. ACM.

[5] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 61–72, 2011.

[6] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. Human-powered sorts and joins. *Proc. VLDB Endow.*, 5(1):13–24, September 2011.

[7] Atsuyuki Morishima, Norihide Shinagawa, Tomomi Mitsuishi, Hideto Aoki, and Shun Fukusumi. Cylog/crowd4u: a declarative platform for complex data-centric crowdsourcing. *Proc. VLDB Endow.*, 5(12):1918–1921, August 2012.

[8] Hyunjung Park, Richard Pang, Aditya Parameswaran, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. An overview of the deco system: data model and query language; query processing and optimization. *SIGMOD Rec.*, 41(4):22–27, January 2013.

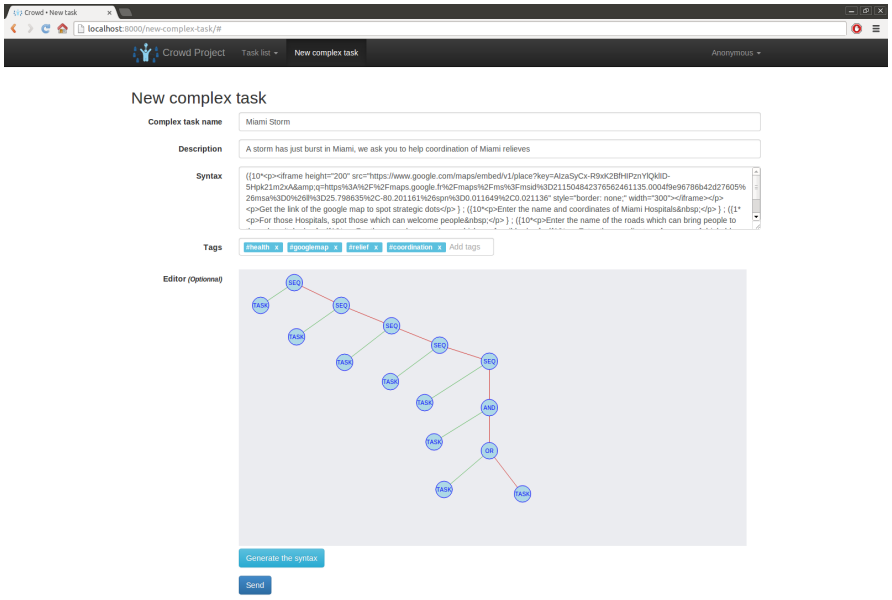


Figure 2: Screenshot of a complex task creation

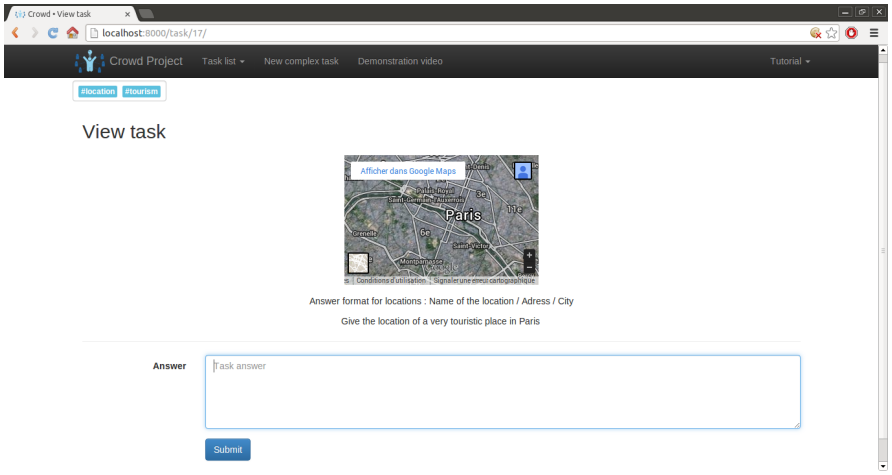


Figure 3: Advanced XHTML features